Lab 3 – Binary Tree

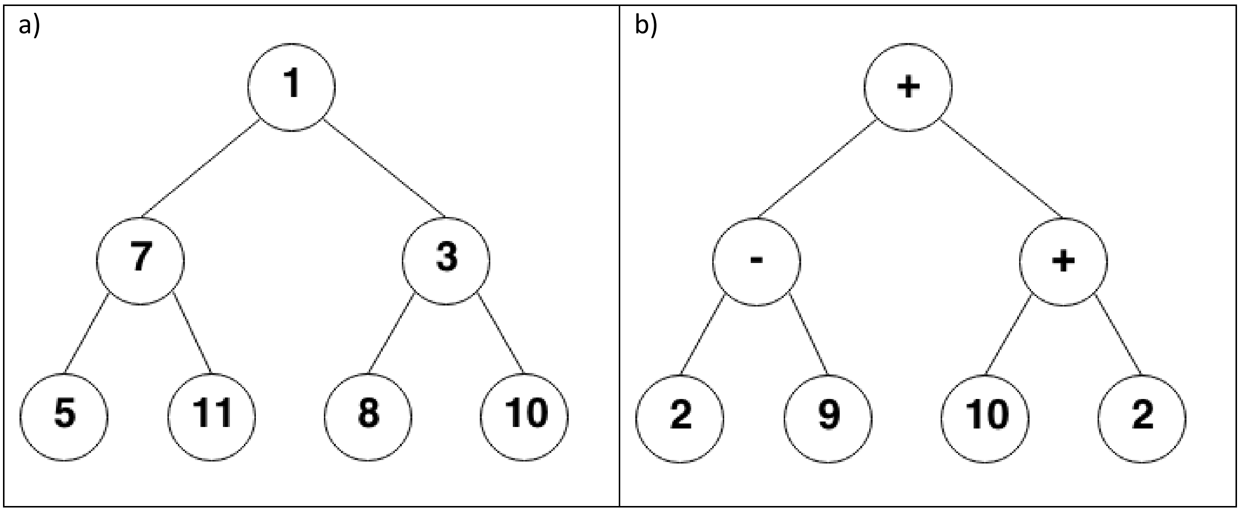
The following code is applied to all questions.

```
#define NODE_TYPE_OPERATOR    0
#define NODE_TYPE_OPERAND     1
#define OPERATOR_ADD     0      // +
#define OPERATOR_MINUS   1      // -

struct NodeEntry
{
      int type;
      int value;
      void printNode() {
            if (type == NODE_TYPE_OPERAND) {
                  cout<<value;
            } else {
                  switch (value) {                    {
                  case OPERATOR_ADD:
                        cout<<"+";
                        break;
                  case OPERATOR_MINUS:
                        cout<<"-";
                        break;
                  }
            }
      }
};

struct TreeNode {
      NodeEntry entry;
      TreeNode *left, *right;

      TreeNode() {      left = right = NULL;      }

      TreeNode(NodeEntry item, TreeNode * left = NULL, TreeNode *
right = NULL)
      {
            this->entry = item;
            this->left = left;
            this->right = right;
      }
};

class BinaryTree {
public:
      BinaryTree(){
            root = NULL;
      }

      ~BinaryTree()
      {
            destroy(root);
            root = NULL;
      }

      bool empty()
      {
```

```
		return (root==NULL);
	}

	bool insertAt(TreeNode *parent, bool leftOrRight, NodeEntry 
data, TreeNode *&newNode)
	{
		if (parent == NULL) {
			if (root != NULL)
				return false;
		} else {
			if ((leftOrRight && (parent->left != NULL))
				|| (!leftOrRight && (parent->right != NULL)))
				return false;
		}

		newNode = new TreeNode(data);
		if (parent == NULL) {
			root = newNode;
		} else {
			if (leftOrRight)
				parent->left = newNode;
			else
				parent->right = newNode;
		}
		return true;
	}

	void printPreOrder()	//NLR
	{
		// add your code here for question 2a
	}

	void printInOrder()	//LNR
	{
		// add your code here for question 2b
	}

	void printPostOrder()
	{
		// add your code here for question 2c
	}

	int heightTree()
	{
		// add your code here for question 3
	}

	int calculateBlanceFactor ()
	{
		//add your code here for question 4
	}

	int countLeaf()
	{
		//add your code here for question 5
	}

	void deleteLeaves()
	{
		//add your code here for question 6
```

```
        }

        TreeNode* findValue(NodeEntry value)
        {
                //add your code here for question 7
        }

        void swapNode()
        {
                //add your code here for question 8
        }

        int caculateTree()
        {
                //add your code here for question 9
        }

        void build_tree_from_keyboard ()
        {
                root = build_tree_from_keyboard_recur();
        }

protected:
        TreeNode *root;

        void destroy(TreeNode * subroot)
        {
                if (subroot != NULL) {
                        destroy(subroot->left);
                        destroy(subroot->right);
                        delete subroot;
                }
        }

        TreeNode * build_tree_from_keyboard_recur ()
        {
                char ans;
                cout << "Enter more (Y/N)? ";
                cin >> ans;
                if (ans == 'Y') {
                        NodeEntry data;
                        cout << "Enter an entry (type, data): \n";
                        cin >> data.type >> data.value;

                        TreeNode * p = new TreeNode(data);
                        cout << "Enter the left sub-tree \n";
                        p->left = build_tree_from_keyboard_recur ();
                        cout << "Enter the right sub-tree \n";
                        p->right = build_tree_from_keyboard_recur ();
                        return p;
                }
                return NULL;
        }
};
```

Question 1. Use the already implemented method ***insertAt*** to construct binary tree as follow:



a)

```
NodeEntry nodeE;
nodeE.type = NODE_TYPE_OPERAND;

TreeNode* rootNode = NULL;
TreeNode* leftFirstNode = NULL;
TreeNode* rightFirstNode = NULL;
TreeNode* freeNode = NULL;

BinaryTree tree;

nodeE.value = 1;
tree.insertAt(NULL,true, nodeE,rootNode);

nodeE.value = 7;
tree.insertAt(rootNode,true, nodeE,leftFirstNode);

nodeE.value = 3;
tree.insertAt(rootNode,false, nodeE,rightFirstNode);

nodeE.value = 5;
tree.insertAt(leftFirstNode,true, nodeE,freeNode);
nodeE.value = 11;
tree.insertAt(leftFirstNode,false, nodeE,freeNode);

nodeE.value = 8;
tree.insertAt(rightFirstNode,true, nodeE,freeNode);
nodeE.value = 10;
tree.insertAt(rightFirstNode,false, nodeE,freeNode);
```

b)

```
NodeEntry nodeE;

TreeNode* rootNode = NULL;
TreeNode* leftFirstNode = NULL;
TreeNode* rightFirstNode = NULL;
TreeNode* freeNode = NULL;
BinaryTree expressionTree;
```

```
nodeE.type = NODE_TYPE_OPERATOR;
nodeE.value = OPERATOR_ADD;
expressionTree.insertAt(NULL,true, nodeE,rootNode);

nodeE.type = NODE_TYPE_OPERATOR;
nodeE.value = OPERATOR_MINUS;
expressionTree.insertAt(rootNode,true, nodeE,leftFirstNode);

nodeE.type = NODE_TYPE_OPERATOR;
nodeE.value = OPERATOR_ADD;
expressionTree.insertAt(rootNode,false, nodeE,rightFirstNode);

nodeE.type = NODE_TYPE_OPERAND;
nodeE.value = 2;
expressionTree.insertAt(leftFirstNode,true, nodeE,freeNode);
nodeE.type = NODE_TYPE_OPERAND;
nodeE.value = 9;
expressionTree.insertAt(leftFirstNode,false, nodeE,freeNode);

nodeE.type = NODE_TYPE_OPERAND;
nodeE.value = 10;
expressionTree.insertAt(rightFirstNode,true, nodeE,freeNode);
nodeE.type = NODE_TYPE_OPERAND;
nodeE.value = 2;
expressionTree.insertAt(rightFirstNode,false, nodeE,freeNode);
```

Question 2. Implement method to print the tree.

a) PreOrder

```
void printPreOrderRecursive(TreeNode* root)
{
    if(root == NULL)
        return;
    root->entry.printNode(); cout << " ";
    printPreOrderRecursive(root->left);
    printPreOrderRecursive(root->right);
}
class BinaryTree  {
…
   void printPreOrder() {  //NLR
        printPreOrderRecursive(this->root);
    }
…
}
```

b) InOrder

```
void printInOrderRecursive(TreeNode* root)
{
    if(root == NULL)
        return;
    printInOrderRecursive(root->left);
    root->entry.printNode(); cout << " ";
    printInOrderRecursive(root->right);
}
class BinaryTree  {
…
   void printInOrder() {   //LNR
        printInOrderRecursive(this->root);
    }
…
}
```

c) PostOrder

```
void printPostOrderRecursive(TreeNode* root) {
    if(root == NULL)
        return;
    printPostOrderRecursive(root->left);
    printPostOrderRecursive(root->right);
    root->entry.printNode(); cout << " ";
}
class BinaryTree  {
…
   void printPostOrder() {
        // add your code here for question 2c
        printPostOrderRecursive(this->root);
    }
…
}
```

Question 3. Implement method ***heightTree*** to calculate the height of the tree.

```
int heightTreeRecursive(TreeNode* root)
{
    if(root==NULL)
        return 0;
    return max(heightTreeRecursive(root->left),
                    heightTreeRecursive(root->right)) + 1;
}
class BinaryTree  {
…
   int heightTree()
    {
        // add your code here for question 3
        return heightTreeRecursive(this->root);
    }
…
}
```


Question 4. Implement method ***calculateBlanceFactor*** to calculate the balance factor of the tree.

```
class BinaryTree  {
…
   int calculateBlanceFactor ()
    {
        //add your code here for question 4
        if(this->root ==NULL) return 0;
        return heightTreeRecursive(this->root->left) -
                    heightTreeRecursive(this->root->right);
    }
…
}
```


Question 5. Implement method ***countLeaf*** to count the number of leaves in the tree.

```
int countLeafRecursive(TreeNode* root) {
    if(root ==NULL)
        return 0;
    if(root->left==NULL&&root->right==NULL)
        return 1;
    return countLeafRecursive(root->left) + countLeafRecursive(root-
>right);
}
```

```
class BinaryTree  {
…
    int countLeaf()
    {
        //add your code here for question 5
        return countLeafRecursive(this->root);
    }
…
}
```

Question 6. Implement method ***deleteLeaves*** to delete all leaves from the tree.

```
void deleteLeavesRecursive(TreeNode*& root)
{
    if(root==NULL) return;
    if(root->left==NULL&&root->right==NULL)
    {
        delete root;
        root = NULL;
        return;
    }
    deleteLeavesRecursive(root->left);
    deleteLeavesRecursive(root->right);
}
class BinaryTree  {
…
    void deleteLeaves()
    {
        //add your code here for question 6
        if(this->root==NULL) return;
        if(this->root->left==NULL&&this->root->right==NULL)
        {
            delete this->root;
            this->root = NULL;
            return;
        }
        deleteLeavesRecursive(this->root->left);
        deleteLeavesRecursive(this->root->right);
    }
…
}
```

Question 7. Implement method ***findValue*** to find a node with value 'value'.

```
TreeNode* findValueRecursive(TreeNode* root,NodeEntry searchEntry) {
    if(root == NULL) return NULL;
    if(root->entry.type==searchEntry.type &&
            root->entry.value==searchEntry.value)
        return root;
    TreeNode* leftResult = findValueRecursive(root->left,searchEntry);
    if(leftResult != NULL) return leftResult;
    return findValueRecursive(root->right,searchEntry);
}
class BinaryTree  {
…
    TreeNode* findValue(NodeEntry value) {
        //add your code here for question 7
        return findValueRecursive(this->root,value);
    }
… }
```

Question 8. Implement method ***swapNode*** to swap the left and the right sub-trees at any node.

```
void swapNodeRecursive(TreeNode* root)
{
    if(root==NULL) return;
    TreeNode* temp = root->left;
    root->left = root->right;
    root->right = temp;
    swapNodeRecursive(root->left);
    swapNodeRecursive(root->right);
}
class BinaryTree  {
…
    void swapNode()
    {
        //add your code here for question 8
        swapNodeRecursive(this->root);
    }
…
}
```

Question 9. Support that an expression can be presented as a binary tree as in question 4.1 (*) In that expression tree, leaves are used to present operands, which are numbers. Other nodes on the tree are used to present operators (plus and minus). Implement method ***caculateTree*** to calculate an expression expressed as a tree above.

```
#define ERROR_CODE -77777 // used for question 9
int caculateTreeRecursive(TreeNode* root) {
    if(root ==NULL) return ERROR_CODE;
    if(root->entry.type == NODE_TYPE_OPERAND) {
        if(root->left!=NULL||root->right!=NULL)
            return ERROR_CODE;
        return root->entry.value;
    }
    int leftResult = caculateTreeRecursive(root->left);
    int rightResult = caculateTreeRecursive(root->right);
    if(leftResult==ERROR_CODE||rightResult==ERROR_CODE) return ERROR_CODE;
    switch (root->entry.value) {
        case OPERATOR_ADD:
            return leftResult+rightResult;
        case OPERATOR_MINUS:
            return leftResult-rightResult;
    }
    return ERROR_CODE;
}
class BinaryTree  {
…
    int caculateTree() {
        //add your code here for question 9
        caculateTreeRecursive(this->root);
    }
…
}
```

(*) Data Structures and Algorithms in C++, 4th, Adam Drozdek, chapter 6, section 6.12, page 286.